

Seminar "XML-Schemata und andere Typen von Graphen"

G-Maschine: Graphersetzung als Basis der Funktionalen Programmierung

Achim Lücking

Sommersemester 2002

Inhaltsverzeichnis

1	Einleitung	2
2	Der λ-Kalkül als Basis funktionaler Programmiersprachen	4
2.1	Syntax des λ -Kalküls	4
2.2	Reduktionsregeln	5
2.3	Darstellung der λ -Ausdrücke als Graph	6
3	Die G-Maschine: ein Auswertungsmodell funktionaler Programme	9
3.1	Die Kombinatorform	9
3.2	Arbeitsweise der G-Maschine	10
4	Erweiterung zur STG-Maschine	14
4.1	Die STG-Sprache	14
4.2	Auswertung mit der STG-Maschine	15
4.2.1	Aufbau eines Zustandes der STG-Maschine	15
4.2.2	Der Anfangszustand	17
4.2.3	Auswertung mit Updates	17
4.2.4	Terminierung	19
5	Zusammenfassung	20

1 Einleitung

Im Gegensatz zu imperativen Programmiersprachen basieren funktionale Sprachen nicht auf Schleifen und Zuweisungen. Stattdessen besteht die Ausführungssemantik im wesentlichen aus Ersetzungs- und Reduktionsregeln, mit denen ein Ausdruck ausgewertet wird. Diese Auswertung kann nach strikten und nach nicht-strikten Schemata durchgeführt werden. Strikte Auswertung bedeutet, dass im innersten Teilterm der am weitesten links stehende Ausdruck als nächstes ausgewertet wird. Als einführendes Beispiel sei eine einfache Funktion zu Berechnung des Quadrates einer Zahl angegeben. Diese Berechnung lässt sich für beliebige Zahlen durch die folgende Termreduktionsgleichung angeben:

$$\text{square}(x) = x*x$$

Die Funktion "*" sei für Zahlen wie gewohnt definiert. Die Auswertung eines Ausdrucks $\text{square}(\text{square}(4))$ beispielsweise ist dann durch Anwendung der Reduktionsregel möglich. Die Auswertung dieses Ausdrucks ist jedoch nicht eindeutig. Der Ausdruck kann dabei

z.B. strikt oder nicht-strikt ausgewertet werden, wie der in Abbildung 1 dargestellte Auswertungsbaum verdeutlicht. Auf Grund der Fülle der Auswertungsmöglichkeiten sei hier nur eine Auswahl der Möglichkeiten angegeben.

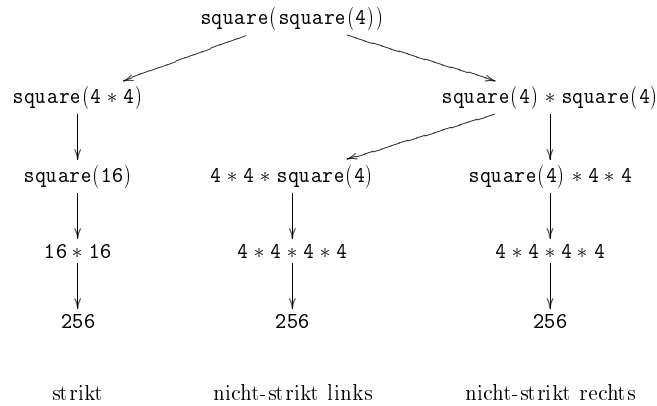


Abbildung 1: Auswertungsbaum des Ausdrucks `square(square(4))`

Die Auswertung eines funktionalen Ausdrucks erfolgt bei funktionalen Compilern in der Regel nach nicht-strikten Schemata. Jeder Knoten des Auswertungsbaums kann wiederum als Termersetzungsgraph angesehen werden. Exemplarisch sei ein nicht-strikter Pfad des Auswertungsbaums mit Hilfe von Termersetzungsgraphen in Abbildung 2 dargestellt. Die

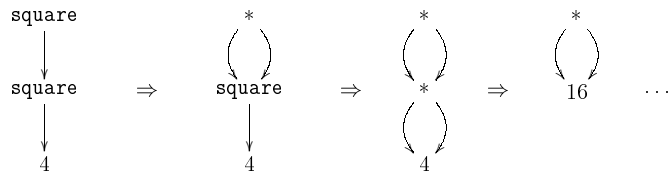


Abbildung 2: Termersetzungsgraphen für einen Auswertungspfad

Auswertungsstrategie auf dem Termersetzungsgraph bestimmt dabei lediglich die Auswertungsstelle der nächsten Regel. Anhand des obigen Beispiels kann man sich vorstellen, dass bei der Auswertung funktionaler Ausdrücke, egal, ob über einen Termersetzungsgraph oder einen abstrakten Syntaxbaum, sehr schnell Graphstrukturen entstehen. Daher lassen sich funktionale Ausdrücke auch mittels Graphreduktion auswerten.

Diese Arbeit führt zunächst in den λ -Kalkül mit seiner Syntax und seinen Reduktions- und Auswertungsregeln ein. Dieser Kalkül stellt die Basis aller funktionalen Programmiersprachen dar. Auf Basis der Ausdrücke dieses Kalküls soll der Aufbau eines Auswertungsgraphen betrachtet werden, der auf dem abstrakten Syntaxbaum aufbaut.

Die so erzeugten Graphen zur Auswertung der funktionalen Ausdrücke dienen als Grundlage der Auswertung mit der sogenannten G-Maschine, die im Kapitel 3 vorgestellt wird

und mit deren Ansatz ein Graph mit imperativen Code ausgewertet und das Ergebnis der Auswertung effizient bestimmt werden kann.

Eine optimiertes G-Maschinen-Konzept wird abschließend in Kapitel 4 betrachtet: die STG-Maschine. Die Erweiterungen und Abgrenzungen zur G-Maschine werden kurz vorgestellt. Außerdem wird kurz auf die Syntax der Maschine eingegangen und der Übergang vom λ -Kalkül zur STG-Sprache gezeigt, auf der die STG-Maschine operiert.

2 Der λ -Kalkül als Basis funktionaler Programmiersprachen

Der λ -Kalkül wurde von Church noch vor allen Programmiersprachen entwickelt. Church nutze ihn als Formalisierung des Begriffs *Berechenbarkeit*. Insbesondere bildet der λ -Kalkül die Grundlage funktionaler Programmiersprachen. Funktionale Programmiersprachen können als syntaktische und vor allem lesbarere Varianten des λ -Kalküls angesehen werden. Jedes funktionale Programm lässt sich demnach in den Kalkül überführen, so dass die einfache Sprache des λ -Kalküls eine vollwertige Programmiersprache darstellt. Im weiteren Verlauf der Arbeit wird mit dem λ -Kalkül der Übergang von funktionalen Sprachen zum Auswertungsgraph für die G-Maschine und der STG-Sprache der STG-Maschine verdeutlicht. Im folgenden seien die Syntax und die Reduktionsregeln kurz vorgestellt.

2.1 Syntax des λ -Kalküls

Die Syntax des λ -Kalküls ist die Sprache der λ -Terme. Die Menge der λ -Terme Λ ist definiert als kleinste Menge für die gilt:

- $\mathcal{C} \subseteq \Lambda$
 \mathcal{C} bezeichne dabei eine Menge von *Konstanten*. Konstanten sind λ -Terme.
- $\mathcal{V} \subseteq \Lambda$
 \mathcal{V} ist eine (abzählbar unendliche) Menge von *Variablen*. Variablen sind λ -Terme.
- $(t_1 t_2) \in \Lambda$, falls gilt: $t_1, t_2 \in \Lambda$
Terme der Form $(t_1 t_2)$ werden auch als *Applikationen* (oder *Anwendungen*) bezeichnet, da sie semantisch gesehen den Term t_1 auf t_2 anwenden.
- $\lambda x.t \in \Lambda$, falls gilt: $x \in \mathcal{V}$ und $t \in \Lambda$
Terme der Form $\lambda x.t$ heißen *Abstraktionen* und repräsentieren anonyme Funktionen. Die Bedeutung eines solchen Terms ist die Funktion, die jeden Wert x auf den Wert von t abbildet.

Zur Vereinfachung der Schreibweise gelten die folgenden Konventionen: Anwendungen assoziieren nach links, so dass der Term $(t_1 t_2 t_3)$ für $((t_1 t_2) t_3)$ steht. Der Wirkungsbereich eines λ erstreckt sich soweit wie möglich nach rechts. Ein Term $\lambda x.\lambda y.t$ wird kurz nur $\lambda x y.t$ geschrieben.

Als Beispiel für einen konkreten λ -Term sei hier das Beispiel `square(x)=x*x` aus der Einleitung noch einmal aufgegriffen. Als λ -Term schreibt man die Gleichung wie folgt: $\lambda x.x * x$

Der λ -Kalkül sieht auch das Konzept der freien Variablen vor, welches auch in vielen funktionalen Sprachen zu finden ist. Eine Variable in einem Term heißt frei, wenn sie nicht durch ein darüber stehendes λ gebunden wird. Ein Term t heißt geschlossen, wenn die Menge der freien Variablen des Terms leer ist. Die Menge der freien Variablen eines λ -Terms kann wie folgt formal beschrieben werden:

- $\text{free}(c) = \emptyset$ für alle $c \in \mathcal{C}$
- $\text{free}(x) = \{x\}$ für alle $x \in \mathcal{V}$
- $\text{free}(t_1 t_2) = \text{free}(t_1) \cup \text{free}(t_2)$ für alle $t_1, t_2 \in \Lambda$
- $\text{free}(\lambda x.t) = \text{free}(t) \setminus \{x\}$ für alle $x \in \mathcal{V}, t \in \Lambda$

Abschließend sei noch die Substitution eingeführt. Eine Substitution ersetzt alle freien Vorkommen einer Variable x durch einen Term t . Die Schreibweise $r[x/t]$ bezeichnet also die Ersetzung von x durch t in einem Term r . Allgemein sei die Substitution für alle $r, t \in \Lambda$ und alle $x \in \mathcal{V}$ wie folgt definiert:

- $y[x/t] = \begin{cases} t & \text{falls } y = x \\ y & \text{falls } y \in \mathcal{V} \cup \mathcal{C}, x \neq y \end{cases}$
- $(r_1 r_2)[x/t] = (r_1[x/t] r_2[x/t])$ für alle $r_1, r_2 \in \Lambda$
- $(\lambda x.r)[x/t] = \lambda x.r$
- $(\lambda y.r)[x/t] = \begin{cases} \lambda y.(r[x/t]) & \text{falls } y \neq x, y \notin \text{free}(t) \\ \lambda y'.(r[y/y'][x/t]) & \text{falls } y \neq x, y \in \text{free}(t), y' \notin \text{free}(t) \end{cases}$

Mit Hilfe dieser Definitionen können nun funktionale Programme beschrieben werden.

2.2 Reduktionsregeln

Um die nach den oben angegebenen Definitionen konstruierten λ -Terme auszuwerten, bedarf es einiger Reduktionsregeln, nach denen bei der Auswertung vorgegangen wird. Im folgende Abschnitt soll einen kurzen Überblick über die wichtigsten Reduktionsregeln gegeben werden.

Die einfachste Form der Reduktion ist die α -Reduktion, auch α -Konversion genannt. Diese Reduktion erlaubt lediglich, gebundene Variablen umzubenennen. Dabei ist darauf zu achten, dass bei der Umbenennung keine Variablen eingeführt werden, die bereits frei im Ausdruck vorkommen. Ein Beispiel für einen Schritt der α -Reduktion ist die folgende Reduktion: $\lambda x y.x * y \rightarrow_\alpha \lambda x z.x * z$

Eine weitere Reduktionsregel ist die β -Reduktion. Sie erlaubt die Anwendung der λ -Abstraktion auf beliebige Terme. Eine λ -Abstraktion $\lambda x.t$ wird auf ein Argument r angewendet. Die Reduktion geschieht dabei durch die Substitution aller Vorkommen der gebundenen Variable x im Rumpf t durch r . Dabei werden die Regeln der Substitution beachtet. Vor der Substitution muss ggf. eine α -Reduktion durchgeführt werden, also eine Umbenennung der gebundenen Variable x , damit es durch die β -Reduktion nicht zu

einer Kollision der Bezeichner kommt. Formal lässt sich die β -Reduktion wie folgt definieren: $(\lambda x.t)r \rightarrow_\beta t[x/r]$. Als konkretes Beispiel sei die folgende β -Reduktion angeführt: $(\lambda x y.x + y) 2 \rightarrow_\beta \lambda y.2 + y$

Eine dritte wichtige Reduktionsregel ist die δ -Reduktion. Hier wird auf einen λ -Teilausdruck eine Regel δ angewendet, die in einer δ -Regelmenge definiert ist. Darunter fallen u.a. Regeln um beispielsweise eine Addition mit dem Funktionssymbol $+$ und zwei Konstanten auszuwerten. Beispiel: $(+ 3 4) \rightarrow_\delta 7$, vorausgesetzt, dass für das Funktionssymbol $+$ eine entsprechende Regel in der zugehörigen δ -Regelmenge definiert ist.

Als für die Graphreduktion mit der STG-Maschine (vgl. Kapitel 4) wichtige Normalform eines λ -Ausdrucks sei noch die Kopf-Normalform eines λ -Ausdrucks definiert. Ein λ -Ausdruck L liegt in Kopf-Normalform vor, wenn gilt:

$$L = \lambda x_1 x \dots x_n.(v M_1 \dots M_m)$$

Dabei ist $m, n \geq 0$, v ein Funktions- oder Datenwert und $(v M_1 \dots M_p), p \leq m$ kein reduzierbarer Ausdruck. Für den Fall $n = 0$ hat L die Form $L = v M_1 \dots M_m$.

2.3 Darstellung der λ -Ausdrücke als Graph

λ -Terme lassen sich alternativ zu ihrer formalen Darstellung auch als Graph angeben, auf dem dann die Auswertung des Ausdrucks in Form einer Graphreduktion vorgenommen werden kann. Dabei stellt dieser Graph am Anfang den abstrakten Syntaxbaum des Ausdrucks dar. Im Zuge der Reduktion entstehen dann "echte" Graphen, die nicht zwingend zyklfrei sind.

Bei der Graphdarstellung unterscheidet man zwischen Blättern, Applikationsknoten und Abstraktionsknoten. In den Blättern werden Variablen, Konstanten und elementare Funktionen dargestellt. Ein Applikationsknoten (er wird mit einem "@" dargestellt) hat zwei Söhne entsprechend der beiden Teilausdrücke. Ein Abstraktionsknoten (er wird mit einem " λ " markiert) hat ebenfalls zwei Söhne: eine Variable, die gebunden wird und einen Rumpf.

Als Beispiel sie hier wieder der Ausdruck `square(x)=x*x` verwendet, welcher als λ -Ausdruck wie folgt dargestellt werden kann: $\lambda x.x * x$. Der zugehörige Graph ist in Abbildung 3 dargestellt. Bei der Auswertung eines Ausdrucks wird nach der nicht-strikten

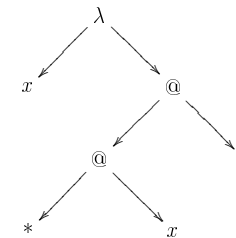


Abbildung 3: Beispielgraph

leftmost-outermost Strategie vorgegangen: den nächsten auszuwertenden Graphknoten

findet man dann, indem man von der Wurzel ausgehend solange nach links geht, bis man auf den ersten Nicht-Applikationsknoten trifft. Dessen Vater (logischerweise ein Applikationsknoten) stellt den nächsten auszuwertenden Redex ("reducible expression") dar. Die "linke Außenkante" des Graphen wird auch als Rückgrat (engl. *spine*) des Graphen bezeichnet. Rekursionen können durch zyklische Graphen dargestellt werden, wobei jedoch auf eine Abbruchbedingung der Rekursion auch im Graphen zu achten ist.

Im folgenden Abschnitt sei nun betrachtet, wie die Reduktionsregeln des λ -Kalküls adäquat auf den Graphen angewendet werden können. Im Falle der α -Reduktion ist dieses trivial, da die Reduktion lediglich eine Umbenennung gebundener Variablen vornimmt. Demnach müssen auch im Graphen die entsprechenden Knoten umbenannt werden.

Die β -Reduktion hingegen ist etwas komplexer. Um die Reduktion im Graphen durchzuführen, muss in einem Redex $(\lambda x.E)A$ jeder x -Knoten in E durch den A -Graphen ersetzt werden. Wegen eines möglichen Mehrfachauftretens von x -Knoten in E wird der A -Graph jedoch nicht einfach kopiert. Man verschmilzt zunächst die x -Knoten, so dass der entstehende Knoten einmal durch A ersetzt wird. Der Applikationsknoten wird abschließend durch den E -Wurzelknoten ersetzt. So wird der Baum auch zu einem "echten" Graphen. Hier erkennt man im Graphen die Strategie der *lazy evaluation*, die besagt, dass ein Teilausdruck erst dann ausgewertet wird, wenn er für die Auswertung des übergeordneten Ausdrucks benötigt wird. Die Auswertung findet jedoch nur ein einziges Mal statt. Diese Auswertungsstrategie wird in vielen funktionalen Sprachen angewendet (z.B. Haskell). Als Beispiel sei folgender λ -Ausdruck gegeben: $(\lambda f.\lambda x.f(f x))(\lambda x.x * x) 4$. Daraus resultiert der Graph aus Abbildung 4. Nach dem beschriebenen Verfahren der β -

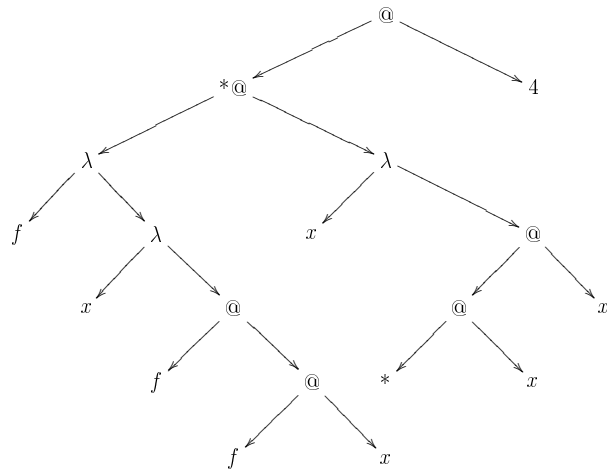


Abbildung 4: Graph des Ausdrucks $(\lambda f.\lambda x.f(f x))(\lambda x.x * x) 4$

Reduktion auf den Graph angewandt auf den mit * gekennzeichneten Applikationsknoten, müssen nun zunächst die beiden f -Knoten im Rumpf verschmolzen werden (Abbildung 5). Anschließend werden sie durch den $(\lambda x.x + 1)$ -Graph ersetzt. Zum Schluss wird der Applikationsknoten durch seinen linken Nachfolger ersetzt. Der Abstraktionsknoten von

f entfällt, da diese Abstraktion in diesem Schritt durchgeführt wurde. Das Ergebnis ist dann der Graph in Abbildung 6.

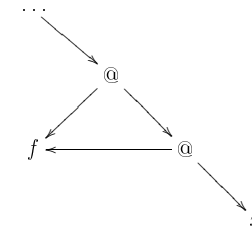


Abbildung 5: Verschmelzen der f -Knoten im Rumpf

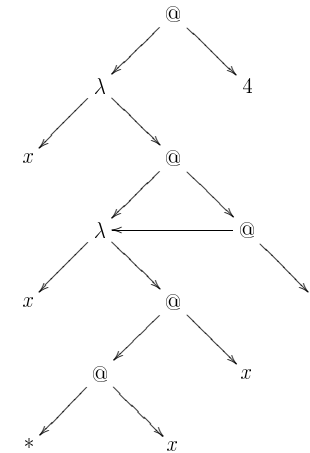


Abbildung 6: Nach β -Reduktion reduzierter Graph

Die Anwendung der δ -Reduktion ist ebenfalls trivial, da gemäß der Regeln aus der δ -Regelmenge ein Redex ausgewertet und im Graph durch das entsprechende Ergebnis ersetzt werden kann. Ein Teilausdruck $(+ 3 1)$ kann im Graph direkt zu 4 ersetzt werden, wenn diese Regel in der δ -Regelmenge vorhanden ist.

Bei der Anwendung und Umsetzung z.B. der β -Reduktion auf einen Ausdruck bzw. dessen Graphen sieht man sehr schnell den großen Nachteil dieser Auswertungsmethode: es können schon für kleine Ausdrücke sehr schnell recht große Auswertungsgraphen entstehen. Es muss also ein effizienterer Ansatz gefunden werden. Eine Möglichkeit wird im folgenden Kapitel vorgestellt.

3 Die G-Maschine: ein Auswertungsmodell funktionaler Programme

Für die Auswertung funktionaler Programme wurden diverse Maschinenmodelle entwickelt. Als Meilenstein unter diesen Maschinenmodellen wird in der Literatur die sogenannte G-Maschine genannt (G steht dabei für Graph). Die Idee der G-Maschine basiert auf der Übersetzung von Ausdrücken des funktionalen Programms in sogenannten G-Code, einen abstrakten Maschinencode. Dieser Code erzeugt bei seiner Ausführung Instanzen der Ausdrücke, so dass eine schnelle Umsetzung der Graphreduktion gewährleistet ist. Auf diese Art und Weise erspart man sich das Durchlaufen des gesamten Syntaxbaums in jeder Instanz des Ausdrucks. Die Maschine durchläuft dann diesen G-Code und liefert so eine effiziente Auswertung des Ausdrucks. Der funktionale Ausdruck muss mit einem G-Maschinen-Compiler in G-Code übersetzt werden.

3.1 Die Kombinatorform

Als Basis der Auswertung mit der G-Maschine dient ein funktionaler Ausdruck, der in der sogenannten Kombinatorform vorliegt. Der Grund, warum man nicht die bisher bekannte Form verwendet, lässt sich an einem Beispiel verdeutlichen. Im vorherigen Kapitel wurde die Umsetzung der Reduktionen des λ -Kalküls auf einen Auswertungsgraphen basierend auf dem abstrakten Syntaxbaum erläutert. Die dort beschriebene β -Reduktion hat einen Nachteil, den man erkennt, wenn man den Term `square(square(4))` aus Kapitel 1 auswertet. Als λ -Term geschrieben ergibt sich der folgende Ausdruck für den Term:

$$(\lambda f x.f(f x)) (\lambda x.x * x) 4$$

Bei der Auswertung nach dem in Kapitel 2 beschriebenen Verfahren entsteht nach nur zwei Reduktionsschritten der Graph aus Abbildung 7. In diesem Graph sieht man, dass für den

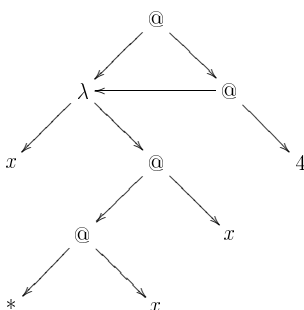


Abbildung 7: Graph nach zwei Reduktionsschritten

nächsten Reduktionsschritt ein Kopie des Rumpfes der Funktion angelegt werden muss, da bei der Auswertung des ersten Redex bereits eine Querverbindung zum zweiten Redex besteht, der λ -Ausdruck also mehrfach verwendet wird. Der Grund liegt im Vorkommen von freien Variablen (in diesem Fall Variablen, die weiter außen gebunden sind). Für

eine korrekte Auswertung muss also eine Kopie des Redex angelegt werden. Um diese ineffiziente Notwendigkeit des Kopierens zu eliminieren, wird bei den auszuwertenden Ausdrücken die Kombinatorform verwendet.

Die Idee der Kombinatorform ist jetzt, freie Variablen zu eliminieren, da diese für obiges Phänomen verantwortlich sind. Man kann nun nach einem Satz aus der kombinatorischen Logik vorgehen, der besagt, dass jeder funktionale Ausdruck in nur drei Kombinatoren nebst Konstanten übersetzt werden kann, wodurch Variablen verschwinden. Dieses Verfahren ist in der Praxis jedoch ungeeignet, da selbst für einfache Funktionen sehr schnell große Kombinatorformen entstehen. Daher verwendet man ein anderes Verfahren, das sogenannte λ -lifting. Dabei wird für jeden λ -Ausdruck, der freie Variablen enthält, ein neuer Kombinator eingeführt, der die freien Variablen als zusätzliche Argumente verwendet. Die freien Variablen werden somit zu Parametern und der Ausdruck so in Kombinatorform überführt. Das λ -lifting betrachtet dabei eine λ -Abstraktion von innen nach außen. Das oben angeführte Beispiel lässt sich in Kombinatorform wie folgt umschreiben:

$$\alpha f x = f(f x) \quad \beta x = * x x \quad \gamma f = \alpha f$$

In dieser Kombinatorform werden uneffiziente Kopien von Teilausdrücken vermieden und die G-Maschine kann einen in dieser Form vorliegenden, funktionalen Ausdruck mit einfachen Reduktionsschritten auswerten.

3.2 Arbeitsweise der G-Maschine

Die G-Maschine verwendet einen einfachen Kellerspeicher (Stack). Auf dem Stack liegen Zeiger auf die Applikationsknoten entlang des Pfades zum Kombinator. Ganz oben auf dem Stack liegt der Zeiger auf den Kombinator selbst. Dieser Pfad ist das bereits im letzten Kapitel angesprochene Rückgrat (engl. *spine*) des Graphen. Man spricht bei der Umsetzung des Rückgrats der Funktion auf den Stack auch vom Abrollen der Funktion. Graphisch kann man sich die Darstellung des Rückgrats auf dem Stack wie in Abbildung 8 vorstellen.

Die Auswertung mit der G-Maschine startet mit einem Zeiger auf die Spitze des Kellerspeichers. Dort kann entweder ein Knoten für einen nullstelligen Kombinator stehen oder ein Applikationsknoten. Wenn es sich um einen nullstelligen Kombinator handelt, so kann sofort die erste seiner Anweisungen angesprungen werden, das Ergebnis der Auswertung kann direkt bestimmt werden. Handelt es sich um einen Applikationsknoten, so hat man zwei Zeiger: einer zeigt auf die anzuwendende Funktion, der andere auf deren Argument. Die anzuwendende Funktion kann wiederum ein Applikationsknoten sein. So wird schrittweise das Rückgrat auf den Stack abgerollt, in dem für jeden Applikationsknoten an der Spitze des Stacks der Zeiger auf die zugehörige Funktion auf den Stack gelegt wird. Der letzte Zeiger, der auf dem Stack abgelegt wird, ist demnach ein Kombinator.

Steht ein Kombinator f an der Stapelspitze, so muss die Stelligkeit von f ermittelt und anschließend die Zahl der Argumente auf dem Stack geprüft werden. Anschließend wird f vom Stapel entfernt, der Stapel umgeordnet und die erste Anweisung von f angesprungen. Beim Umordnen werden Zeiger auf die Anwendungsknoten durch Zeiger auf die Ausdrücke, die Argumente für f sind, ersetzt. Die Maschine fährt anschließend mit den Anweisungen von f fort. Die Anweisungen der Kombinatoren und somit auch von f liegen bereits durch den G-Maschinen-Compiler in Maschinencode vor, so dass diese

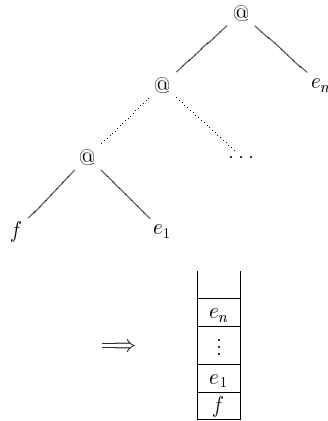


Abbildung 8: Rückgrat eines Graphen und Stack-Darstellung

direkt ausgeführt werden können. Die benötigten Argumente liegen ja bereits auf dem Stack.

Diese Vorgehensweise kann rekursiv auf auszuwertende Teilausdrücke angewendet werden. Auch hier wird an der Wurzel des Redex angesetzt und über die Applikationsknoten das Rückgrat des Teilausdrucks abgerollt.

Die gesamte Auswertung kann man in einer Hauptschleife in folgendem Algorithmus grob zusammenfassen:

- 1.) bestimme den nächsten Redex (mit Hilfe des Kellerspeichers für das Rückgrat des betrachteten Ausdrucks)
- 2.) ist der Redex ein Kombinatorredex, dann wende den Maschinencode für den Kombinator an (wodurch der Graph manipuliert wird), wenn der Redex hingegen eine Applikationsredex ist, dann werte die Argumente (ggf. rekursiv) aus, berechne den Funktionswert und ersetze den Redex durch das Resultat
- 3.) aktualisiere den Kellerspeicher mit dem Rückgrat

Zur Verdeutlichung sei noch einmal das Beispiel vom Anfang des Kapitels herangezogen. Der Ausdruck `square(square(4))` lässt sich, wie bereits erläutert, in Kombinatorform wie folgt angeben:

$$\alpha f x = f(f x) \quad \beta x = * x x \quad \gamma f = \alpha f$$

Zu jedem Kombinator gibt es nun eine Graphreduktionsregel, wie sie in Abbildung 9 abgebildet sind. Diese entstehen quasi durch Umsetzung der linken und der rechten Seite der Regelgleichung. Um nun den Ausdruck `square(square(4))` auszuwerten, muss der Graph für $\gamma \beta 4$ reduziert werden. Dabei wird nach dem obigen Algorithmus vorgegangen.

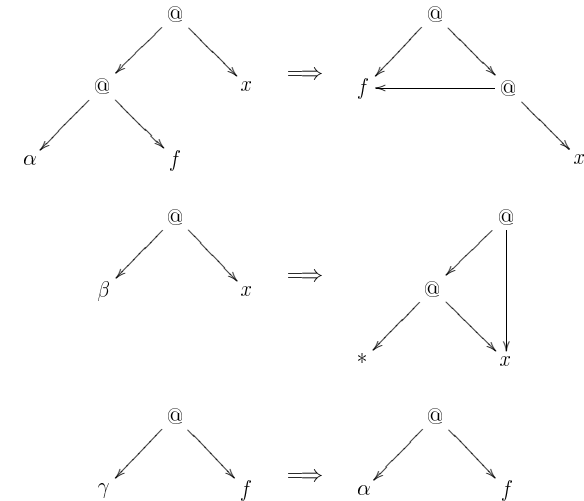


Abbildung 9: Graphreduktionsregeln für $\alpha f x$, βx und γf

Auf die Angabe des jeweiligen Stack für jeden Schritt werde ich an dieser Stelle verzichten. Die Auswertung des Graphen zeigt Abbildung 10.

Man erkennt in der Abbildung, dass der Graph effizient ausgewertet wird, da unnötige Kopien vermieden werden und der Graph während der Auswertung sehr kompakt bleibt. Nach dem Verfahren der β -Reduktion auf dem abstrakten Syntaxbaum wäre der Graph ungleich größer geworden, nicht zuletzt durch notwendige Kopien von Teilausdrücken.

Betrachtet man das Auswertungsverfahren der G-Maschine, so stellt man fest, dass bei der Auswertung eines funktionalen Ausdrucks in jedem Schritt nichts anderes passiert, als das Umlegen von Zeigern, das Anlegen von Applikationsknoten und die Ausführung von δ -Reduktionen über elementare Operationen. Die dazu notwendigen Maschinenbefehle werden vom G-Maschinen-Compiler für jeden Kombinator erzeugt. Außerdem erzeugt der Compiler Befehle, um den anfänglichen Graphen aufzubauen. In Ermangelung eines Compilers (die G-Maschine in ihrer Reinform wird heute leider nirgendwo mehr verwendet), kann an dieser Stelle kein konkreter G-Code für das Beispiel angegeben werden. Ein ausführlicheres Beispiel der Auswertung eines funktionalen Ausdrucks (Fibonacci-Zahlen) findet sich bei [Sch94]. Dort ist auch der konkrete G-Code und die komplette Graphreduktion für das Beispiel mit angegeben.

Die so erzeugte Graphreduktion stellt damit eine effizientere Möglichkeit der Auswertung da. Es gibt weitere Optimierungen dieses Konzeptes, z.B. die Verwendung von Extra-Stacks oder die Überführung von Teilausdrücken in Stack-Code. Ein weiter optimiertes Konzept stellt das nächste Kapitel mit der STG-Maschine vor.

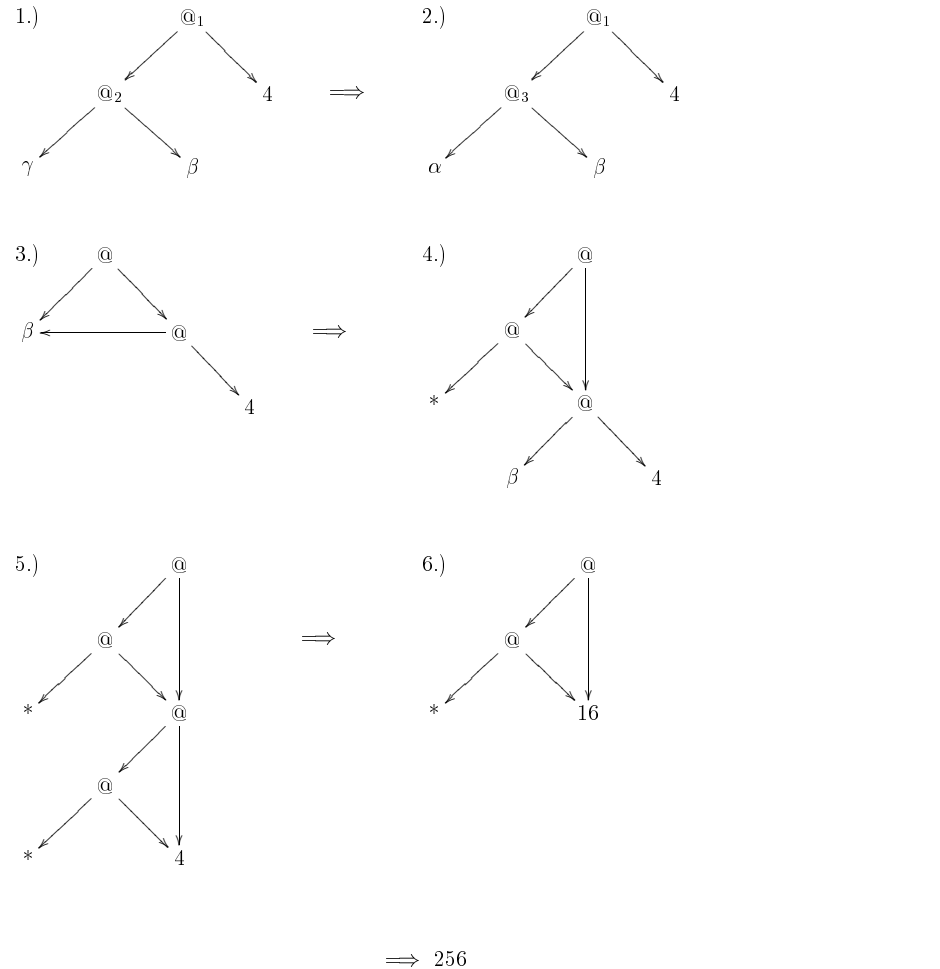


Abbildung 10: Graphreduktion auf dem Ausdruck $\gamma \beta 4$

4 Erweiterung zur STG-Maschine

Eine Erweiterung der G-Maschine ist die Spineless Tagless G-Maschine (kurz: STG-Maschine). Auch hier liegt wieder eine Graphreduktionsmaschine vor, die nun aber ein besonderes Augenmerk auf die effiziente Einzelschrittsemantik legt. Bislang wurden bei der G-Maschine Argumente einzeln einer Funktion übergeben und es entstanden so stets neue Funktionen auf dem Heap. Die STG-Maschine hat, wie der Name schon sagt, kein Rückgrat mehr (daher: spineless). Anwendungen von Funktionen können ohne zusätzliche Heap-Speicherzellen ausgewertet werden. Ferner ist die STG-Maschine in der Lage, in einem Schritt beliebig viele Argumente auf eine Funktion anzuwenden.

Ein weiterer Unterschied zur G-Maschine ist die Eigenschaft, dass die Werte auf dem Heap eine einheitliche Darstellung haben, so dass keine Markierung (engl. *tag*) mehr mitgespeichert werden muss. Bei der G-Maschine musste für jedes Element auf dem Heap unterschieden werden, um was für eine Art Wert (Funktions-/Datenwert oder noch auszuwertender Ausdruck) es sich handelt. Der Nachteil der G-Maschine ist, dass in jedem Schritt erst einmal die Markierung ausgewertet werden muss. Dieser Nachteil wird durch die einheitliche Darstellung bei der STG-Maschine behoben. Hier sind keine interpretierenden Zwischenschritte notwendig, da jede Speicherzelle eine "Methode" beinhaltet, die direkt angesprungen werden kann.

Der Hauptunterschied liegt neben den bereits erwähnten besonderen Merkmalen in der Maschinensprache, die die STG-Maschine interpretiert. Während die G-Maschine noch auf G-Code, also einem abstrakten Maschinencode, operiert, verwendet die STG-Maschine eine kleine funktionale Zwischensprache, die STG-Sprache, auf der sie operiert. Das hat den Vorteil, dass die Portierung von einer höheren funktionalen Programmiersprache in die STG-Sprache nicht so komplex ist, wie die Übersetzung in G-Code. Die STG-Sprache besteht aus einer speziellen Notation von λ -Ausdrücken, so dass eine Umsetzung vom λ -Kalkül in die STG-Sprache keinen großen Aufwand darstellt.

4.1 Die STG-Sprache

Die funktionale Zwischensprache der STG-Maschine, die STG-Sprache, basiert im wesentlichen auf λ -Ausdrücken. Diese Ausdrücke fallen im Gegensatz zu der bereits bekannten Form eines λ -Ausdrucks durch eine spezielle Notation auf. Ein λ -Ausdruck in der STG-Sprache definiert sich in der folgenden Form:

$$l = \{v_1, \dots, v_n\} \backslash \pi \{x_1, \dots, x_m\} \rightarrow expr$$

Dabei sind $\{v_1, \dots, v_n\}$ die freien Variablen des Ausdrucks und $\{x_1, \dots, x_m\}$ die Parameter der Funktion. π ist ein sogenanntes Updateflag, welches später bei der Auswertung näher beschrieben wird. Man sieht hier schnell, dass ein Übergang vom λ -Kalkül zur λ -Form der STG-Sprache nur eine formale Umstellung bedeutet. Eine Übersetzung kann zum Beispiel wie folgt aussehen:

$$(\lambda x. x * x) \quad \Longrightarrow \quad \{\} \backslash \pi \{x\} \rightarrow * \# \{x, x\}$$

Die gesamte Syntax der STG-Sprache lässt sich zusammenfassend formal wie in Abbildung 11 angeben. Ein STG-Programm besteht also vom Grundprinzip her aus einer Menge von Bindungen. Der Wert des gesamten Programms ist der Wert der auf diese Weise

Programm	$prog$	$\rightarrow binds$	
Bindungen	$binds$	$\rightarrow var_1 = lf_1 : \dots : var_n = lf_n$	$n \geq 1$
λ -Ausdr.	lf	$\rightarrow vars_f \setminus \pi vars_a \rightarrow expr$	
Update-Flag	π	$\rightarrow u$	updaten
		$ n$	nicht upd.
Ausdruck	$expr$	$\rightarrow let binds in expr$	lokale Def.
		$ letrec binds in expr$	lokale Rek.
		$ case expr of alts$	Case-Ausdr.
		$ var atoms$	Fkt.appl.
		$ constr atoms$	Konstr.Anw.
		$ prim atoms$	
		$ literal$	
Alternativen	$alts$	$\rightarrow aalt_1; \dots; aalt_n; default$	$n \geq 0$ alg.
		$ palt_1; \dots; palt_n; default$	$n \geq 0$ prim.
alg. Altern.	$aalt$	$\rightarrow constr vars \rightarrow expr$	
prim. Altern.	$palt$	$\rightarrow literal \rightarrow expr$	
Def. Altern.	$default$	$\rightarrow var \rightarrow expr$	
		$ default \rightarrow expr$	
Literale	$literal$	$\rightarrow 0\# 1\# 2\# 3\# \dots$	Int.zahlen
Prim. Oper.	$prim$	$\rightarrow +\# - \# * \# / \# \dots$	Int.oper.
Var.liste	$vars$	$\rightarrow \{var_1, \dots, var_n\}$	$n \geq 0$
Atom-Liste	$atoms$	$\rightarrow \{atom_1, \dots, atom_n\}$	$n \geq 0$
	$atom$	$\rightarrow var literal$	

Abbildung 11: Syntax der STG-Sprache

gebundenen parameterlosen Funktion `main`. Für die Auswertung des Beispiel `square` (4) kann das Programm in STG-Sprache wie folgt angegeben werden:

```
main = {} \n {} -> square {4}:
square = {} \n {x} -> *#\ {x,x}
```

4.2 Auswertung mit der STG-Maschine

Die Auswertung von Programmen kann über die operationelle Semantik beschrieben werden. Da man eine operationelle Semantik sehr gut durch ein Transitionssystem darstellen kann, gibt man für die STG-Maschine einen Anfangszustand und eine Reihe von Transformationsregeln an. Ein Zustand der Maschine wird über ein 6-Tupel dargestellt. Das 6-Tupel besteht aus einer Code-Komponente, drei Stacks, dem Heap und einer globalen Umgebung.

4.2.1 Aufbau eines Zustandes der STG-Maschine

Der Heap versteht sich als Abbildung von Adressen auf Speicherzellen, deren Inhalt als *Closure* bezeichnet wird. Jede Closure kann als Funktion und somit als geschlossener Ausdruck (daher der Name) interpretiert werden. Der Inhalt der Closure besteht aus

einem λ -Ausdruck, wie in der Sprache bereits definiert, und Werten, wobei ein Wert entweder eine weitere Heapadresse oder eine Zahl ist. Über die Werte werden die freien Variablen des λ -Ausdrucks belegt. Es liegt immer die gleiche Anzahl von Werten und freien Variablen vor.

In der Code-Komponente sind die Informationen über die nächste Aktion der STG-Maschine enthalten. Sie nimmt einen von vier Zuständen an, die einem Befehl entsprechen, der ausgeführt wird. Die folgenden vier Befehle sind möglich:

- **Eval** *Ausdruck* ρ
Wertet den Ausdruck unter Berücksichtigung der lokalen Umgebung ρ aus und wendet seinen Wert auf die Argumente auf dem Argument-Stack an.
- **Enter** *Heapadresse*
Wendet die Closure an der gegebenen Heap-Adresse auf die Argumente auf dem Argument-Stack an. Es erfolgt dabei eine Bindung der freien Variablen des λ -Ausdrucks an die Werte aus dem Closure und eine Bindung der gebundenen Variablen an Werte vom Argument-Stack. Das Ergebnis ist eine lokale Umgebung für **Eval**.
- **ReturnInt** *Zahl*
Gibt eine Zahl an die aufrufende Berechnung, also eine Fortsetzung vom Update-Stack, zurück. Bei einem Case-Ausdruck wird dann die passende Alternative gewählt und deren Auswertung initiiert. Im anderen Fall ist die Zahl das Endergebnis einer Auswertung.
- **ReturnCon** *Konstruktor Werte*
Wendet den Konstruktor auf die Werte an und gibt das entstehende Datenobjekt zurück an eine Fortsetzung, die auf dem Update-Stack abgelegt wurde oder terminiert.

Die Übergänge von einem Zustand der Code-Komponente in einen anderen kann über einen endlichen Automaten über die möglichen Befehle wie in Abbildung 12 dargestellt werden. Bei den Schleifen an den Befehlen **Enter** und **ReturnCon** erfolgt jeweils ein sogenanntes Update. Bei einem Update werden dieselben Zustände erneut durchlaufen, jedoch mit zuvor nach bestimmten, vom aktuellen Zustand abhängigen Regeln geänderten Zuordnungen für freie und gebundene Variablen oder Ausdrücke auf dem Heap an der entsprechenden Adresse.

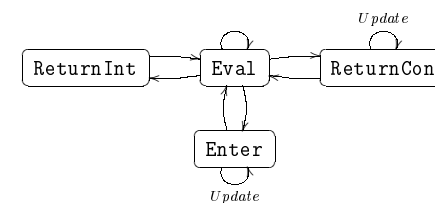


Abbildung 12: Zustandsübergänge in der Code-Komponente

Die drei Stacks der STG-Maschine sind der Argument-Stack, der Return-Stack und der Update-Stack.

- **Argument-Stack**

Der Argument-Stack nimmt, wie der Name bereits sagt, Argumente für Funktionsaufrufe auf. Diese Argumente können entweder Heapadressen oder Integerzahlen sein. Der Stack wird bei der Auswertung von Funktionsapplikationen aufgebaut und beim Betreten von Closures gelesen.

- **Return-Stack**

Der Return-Stack wird für die korrekte Auswertung von Case-Ausdrücken benötigt. Bei der Auswertung eines solchen Ausdrucks werden die Alternativen und die aktuellen Variablenbelegungen (auch *Fortsetzung* oder *Continuations* genannt) auf dem Return-Stack gesichert. Anschließend wird mit der Auswertung des Teilausdrucks fortgefahren. Nach Abschluss der Auswertung wird der Stack ausgelesen und die passende Alternative ausgewählt.

- **Update-Stack**

Der Update-Stack wird nur bei einem gesetzten Update-Flag $\backslash u$ eines λ -Ausdrucks verwendet. Wird eine Closure mit diesem Flag betreten, wird ein Eintrag mit der entsprechenden Adresse auf dem Stack abgelegt. Zusätzlich werden sowohl der Return-Stack als auch der Argument-Stack auf dem Update-Stack gesichert und entleert. Die Auswertung eines Closure startet also mit leeren Stacks. Daher wird später bei einem leeren Return-Stack ein Update ausgelöst. Bei dem Update wird der oberste Stack-Eintrag ausgelesen und die anderen Stack wiederhergestellt. Die Closure an der gespeicherten Adresse wird durch das Auswertungsergebnis überschrieben.

In der STG-Maschine werden auch noch Ausdrücke mit Variablen verwendet, die es erforderlich machen, eine globale Umgebung zu verwenden. Die globale Umgebung ordnet jeder global im Programm definierten Variable die zugehörige Adresse auf dem Heap zu, an der das entsprechende Closure zu finden ist. Die globale Umgebung ist im Gegensatz zu den anderen Komponenten konstant und wird im Laufe der Auswertung nicht verändert.

4.2.2 Der Anfangszustand

Für die Auswertung eines funktionalen Programms startet die STG-Maschine in einem Anfangszustand. Im Anfangszustand sind alle drei Stacks zunächst leer, da es weder Argumente, noch wartende Case-Anweisungen oder Updates gibt. Die Heapbelegung ergibt sich aus den Top-Level-Bindungen, in deren belegten Heapzellen die λ -Ausdrücke abgelegt werden. Die lokale Umgebung enthält die Adressen der Closures, die den Variablen der Bindungen zugeordnet sind. Diese lokale Umgebung ist zugleich auch die *globale* Umgebung des Programms. Die Code-Komponente enthält nur einen Befehl, der *Eval* (`main {}`) lautet, d.h. die Abarbeitung startet mit der Auswertung der Funktion `main` (vgl. [Jon92]).

4.2.3 Auswertung mit Updates

Funktionsaufrufe werden bei der STG-Maschine dadurch realisiert, dass zunächst alle Elemente auf den Stack gelegt werden und dann der Code der Funktion angesprungen wird. Man nennt diese Strategie auch *Push-Enter-Verfahren*. Für die gesamte Übersicht der

Funktionsaufrufe verweise ich auf die Literatur¹. Die Angabe der kompletten Übergangsregeln würde den Rahmen dieser Arbeit sprengen.

Im Zustand *Eval* der Code-Komponente erfolgt, wie bereits kurz beschrieben, die Auswertung des angegebenen funktionalen Ausdrucks. Dabei werden mehrere Fallunterscheidungen gemäß der Syntax des Ausdrucks getroffen und ggf. auftretende Unterausdrücke über einen Zustandswechsel in *Eval Unterausdruck* weiter ausgewertet. Ist im auszuwertenden Ausdruck eine Variablen-Applikation an eine Closure und damit an eine Funktion gebunden (die Variable beinhaltet eine Adresse), so erfolgt ein Zustandswechsel zu *Enter*, die Closure wird also "betreten" und die Funktion ausgeführt. Zuvor werden noch die Werte der aktuellen Umgebung auf den Argument-Stack geschrieben.

Für Berechnung einer Closure wird nun das Update-Flag benötigt. Um zu entscheiden, wann welches Flag gesetzt werden muss ($\backslash u$ für Update oder $\backslash n$ für kein Update), können verschiedene Klassen von λ -Ausdrücken gebildet werden.

- **Manifeste Funktionen**

Manifeste Funktionen sind Funktionen mit nicht leerer Parameterliste. Hier macht ein Update keinen Sinn, da die Funktion bereits in Kopf-Normalform vorliegt und nicht weiter reduziert wird.

- **Partielle Aufrufe**

Ein partieller Aufruf hat die Form

$$vs \backslash n \{ \} \rightarrow f \{ x_1, \dots, x_m \}$$

Dabei ist f eine Funktion, die mehr als m Parameter benötigt. Da aber auch hier die Kopf-Normalform vorliegt, wird das Updateflag auf $\backslash n$ gesetzt.

- **Konstruktorausdruck**

Ein Konstruktorausdruck hat die Form

$$vs \backslash n \{ \} \rightarrow c \{ x_1, \dots, x_m \}$$

Ein Konstruktorausdruck hat immer das Updateflag $\backslash n$.

- **verzögerte Auswertungen**

Einzig bei verzögerten Auswertungen wird das Updateflag auf $\backslash u$ gesetzt. Wird eine verzögerte Auswertung mindestens zweimal benötigt, erreicht man einen Effizienzgewinn.

Diese Flags kommen nun beim Betreten einer Closure mit *Enter Adresse* zum tragen. Wenn das Flag auf $\backslash n$ gesetzt ist, also kein Update erfolgen soll, wird versucht, für jede gebundene Variable des Ausdrucks eine Belegung vom Argument-Stack zu lesen. Wenn dieses gelingt, liegt eine neue Umgebung ρ vor, in der alle gebundenen Variablen eindeutig belegt sind, und es kann mit der Auswertung des Ausdrucks über *Eval Ausdruck* ρ fortgefahren werden. Der Argument-Stack ist um die gelesenen Argumente reduziert worden. Wenn zu wenig Argumente als benötigt auf dem Stack liegen, dann gibt es zwei mögliche Ursachen: ein Typfehler im Programm oder ein Update einer partiellen Applikation. Im letzteren Fall ist folgendes passiert: bei der Auswertung einer partiellen Applikation lagen zu wenig Argumente auf dem Argument-Stack. Deswegen wird der aktuelle Argument-Stack

¹eine Kurzübersicht wird in [Jon92], [Nie98] und [Wel00] gegeben

mit dem gesicherten Argument-Stack vom Update-Stack verschmolzen, damit genügend Argumente vorliegen, und ein Heapzellen-Update durchgeführt. Anschließend wird das Update durch erneutes Durchlaufen des aktuellen Zustandes mit dem neuen Heap und Argument-Stack durchgeführt.

Bei gesetztem Update-Flag soll die Closure nach der Berechnung ihres Inhaltes durch das Resultat ersetzt werden. Dazu wird zunächst der Argument- und Return-Stack auf den Update-Stack geschrieben und beide Stacks geleert, um eine weitere Verwendung übergeordneter Argumente zu verhindern. Durch den nun leeren Return-Stack wird ein Update der Heapzelle ausgelöst und dann das Resultat des ausgewerteten Ausdrucks an die aufrufende Berechnung zurückgegeben.

Im Zustand `ReturnCon` sind eine Reihe von Unterscheidungen zu treffen. Bei leerem Return-Stack ist dieses im aktuellen Ausdruck die letzte Rückgabe. Ist zusätzlich der Update-Stack leer, so terminiert das Programm, im anderen Fall kann ein Update durchgeführt werden. Das Resultat wird also als neue Closure in die Heapzelle mit der auf dem Update-Stack liegenden Adresse geschrieben. Um aus dem Resultat die Closure zu erzeugen, werden freie Variablen benutzt, die mit den übergebenen Werten belegt werden. Der Return- und Update-Stack werden anschließend wieder hergestellt. Abschließend wird unter der geänderten Umgebung der aktuelle Zustand noch einmal durchlaufen und damit das Update durchgeführt. Bei nicht leerem Return-Stack liegen auf dem Stack in der Regel Alternativen eines Case-Ausdrucks. Kann der Konstruktor einer Alternative zugeordnet werden, wird der zugehörige Ausdruck weiter ausgewertet, gleiches gilt für die Standardalternative, falls keine Alternative gefunden werden konnte.

Der letzte Zustand, der noch etwas näher erläutert werden soll, ist der Zustand `ReturnInt`. Hier ist lediglich zu unterscheiden, ob der Return-Stack leer ist oder nicht. Im ersten Fall liegt eine Termination des Programms vor. Im zweiten Fall liegen im Normalfall auf dem Return-Stack die Alternativen eines zuvor ausgewerteten Case-Ausdrucks. Trifft die Zahl auf eine der Alternativen zu, so wird der entsprechend zugehörige Ausdruck mit der aktuellen Umgebung weiter ausgewertet, genauso, wenn die Zahl auf keine der Alternativen zutrifft, aber eine Standardalternative mit `default` existiert.

Eine sehr gute graphische Übersicht über die Vorgehensweise der Auswertung bei den einzelnen Zuständen der Code-Komponente findet sich bei [Wel00] in Kapitel 3. Ein konkretes Auswertungsbeispiel für die STG-Maschine ist in [Nie98] angegeben.

4.2.4 Terminierung

Die Auswertung eines Ausdrucks durch die Maschine ist dann beendet, wenn keine Auswertungsregel mehr anwendbar ist, also kein Folgezustand angegeben werden kann oder der Ausdruck in Kopf-Normalform vorliegt (vgl. [Jon92]). Man spricht dann davon, dass die Maschine terminiert. Mehrere Gründe können für die Terminierung der Maschine zuständig sein:

- Korrekte Berechnung
Das Programm wurde komplett ausgewertet, das Ergebnis steht in der Code-Komponente.
- Unvollständige Berechnung
Die STG-Maschine realisiert eine verzögerte Auswertung. Da die Maschine bereits

nach Berechnung der Kopf-Normalform terminiert, kann das Ergebnis unberechnete Teilausdrücke enthalten.

- Nicht korrektes STG-Programm
Bei von Hand generierten STG-Programmen kann es vorkommen, dass zur Laufzeit Typfehler oder Ähnliches auftreten. Dies kann zu unvorhergesehenen Maschinenzuständen führen. Bei einer Generierung des STG-Codes mit einem Compiler ist dieses in der Regel ausgeschlossen, da der Compiler zuvor bereits Typprüfungen und Ähnliches vornimmt.
- Physikalische Probleme
Bei der Auswertung ist der hardwaremäßig gegebene Speicher für den Aufbau der Stacks und des Heaps begrenzt. Unter Umständen kann es zu einem Überlauf gekommen sein.

Ein korrekter Terminierungszustand ist also nur in den ersten beiden Fällen gemäß den Spezifikationen in [Jon92] gegeben.

5 Zusammenfassung

In dieser Arbeit sind zwei Konzepte vorgestellt worden, um funktionale Ausdrücke mittels Graphreduktion auszuwerten. Dieses Verfahren findet sich heute in vielen funktionalen Compilern. Die STG-Maschine wird in der Praxis beispielsweise im Glasgow Haskell Compiler (GHC) verwendet, um imperativen Code aus einem funktionalen Haskell-Programm zu erzeugen, welcher dann zum fertigen Programm weitercompiliert wird. Obwohl der Compiler auf der STG-Maschine beruht, kann man dort bereits differentiellen Code zur in [Jon92] angegebenen Spezifikation feststellen, vermutlich auch schon aufgrund weiterer Verbesserungen und Anpassungen des Maschinenmodells. Auch eine Erweiterung um die Konzepte des Memoization und Caching (vgl. auch [Wel00]) stellen interessante Erweiterungen bei der STG-Maschine dar, da sich so wiederholte identische Berechnungen vermeiden lassen.

Insgesamt zeigt sich, dass die G-Maschine und ihre optimierten und erweiterten Nachfolger ein effizientes Werkzeug zur imperativen Auswertung eines funktionalen Ausdrucks darstellen. In der Literatur finden sich noch Hinweise, dass die G-Maschine derzeit noch etwa halb so schnell sein soll, wie ein entsprechendes C-Programm, doch wird dort auch damit gerechnet, dass bei weiterer Entwicklung ein Gleichstand bald erreicht sein wird. Durch stetige Optimierungen ist die Graphreduktion so zu einem Werkzeug mit realistischer Effizienz bei der Auswertung funktionaler Ausdrücke geworden.

Literatur

- [Car83] Luca Cardelli. *The Functional Abstract Machine*. AT&T Laboratories Technical Report, Murray Hill, 1983.
- [Gie01] Prof. Dr. Jürgen Giesl. *Grundlagen der funktionalen Programmierung*. Skript zur Vorlesung an der RWTH Aachen, Sommersemester 2001, Aachen, 2001.

- [Joh92] Thomas Johnsson. *Target Code Generation From G-machine Code*. Glasgow, 1992.
- [Jon92] Simon L. Peyton Jones. *Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-Machine*. Journal of Functional Programming 2(2): 127-202, 1992.
- [Kre96] Hubert Kreten. *Graphalgorithmen in PROGRES - Auswahl, Spezifikation und effiziente Realisierung*. Diplomarbeit, RWTH Aachen, Aachen, 1996.
- [Nie98] Philipp Niederau. *Die STG-Maschine*. Seminar "Implementierung deklarativer Programmiersprachen", RWTH Aachen, WS 1997/98.
- [Sch94] Marko Schütz. *Striktheitsanalyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache*. Frankfurt am Main, 1994.
- [Wel00] Ralf Welter. *Simulation einer STG-Maschine mit nicht-striktem Cache*. Diplomarbeit, RWTH Aachen, Aachen, 2000.

Abbildungsverzeichnis

1	Auswertungsbaum des Ausdrucks <code>square(square(4))</code>	3
2	Termersetzungsgraphen für einen Auswertungspfad	3
3	Beispielgraph	6
4	Graph des Ausdrucks $(\lambda f. \lambda x. f(f\ x))(\lambda x. x * x)$ 4	7
5	Verschmelzen der f -Knoten im Rumpf	8
6	Nach β -Reduktion reduzierter Graph	8
7	Graph nach zwei Reduktionsschritten	9
8	Rückgrat eines Graphen und Stack-Darstellung	11
9	Graphreduktionsregeln für $\alpha\ f\ x$, $\beta\ x$ und $\gamma\ f$	12
10	Graphreduktion auf dem Ausdruck $\gamma\ \beta\ 4$	13
11	Syntax der STG-Sprache	15
12	Zustandsübergänge in der Code-Komponente	16